

CSC 310 - Imperative Programming Languages, Spring, 2010, Dr. Dale E. Parson

Assignment 4, emulating **coroutines** using Python **generators** (functions that yield their values iteratively using the Python **yield** statement rather than returning values), modeled after a Scheme program that emulates coroutines using Scheme **continuations**.

Assignment is due at 11:59 PM on April 16, 2010. Turn it in using **gmake turnitin** as usual.

Perform these steps to access the initial source, which is given below.

```
cp ~parson/ProcLang/coroutines_assn4.zip ~/ProcLang
cd ~/ProcLang
/bin/unzip coroutines_assn4.zip
cd ./coroutines
gmake clean test
gmake testscheme
```

Running **gmake test** does nothing at present because you must write the test driver and **makefile** test invocations are part of this assignment. Invoking **gmake testscheme** runs a Scheme program in an infinite loop, which you must break using control-C. Scheme program **producer-consumer.scm** emulates **coroutines** using Scheme **continuations**. It's listing appears below; we will go over it in class. Your program in **producer-consumer.py** must implement the same producer-consumer application logic, emulating **coroutines** using two Python **generators**, which are function-like constructs that produce values iteratively using the **yield** statement. **YOUR PRODUCER AND CONSUMER MUST BE PYTHON GENERATORS THAT USE THE yield COMMAND. IMPLEMENTING THIS PROJECT ANY OTHER WAY IS INCORRECT.** See **producer-consumer.py** below for detailed student requirements.

ProcLang/coroutines/producer-consumer.scm

```
1  ;; Producer-Consumer problem as recursive functions
2  ;; that take each other's continuation as a parameter
3
4  ;(define call/cc call-with-current-continuation)
5
6  (define buffer '()) ;the shared resource
7
8  (define producer
9    (lambda (consumer-continuation count)
10     (set! buffer (list "tuna sandwich #" count)) ;accessing the shared resource
11     (display "producer making ")
12     (display buffer)
13     (newline)
14     ;;(sleep 1) ;slowing things down a bit
15     (foreign-code "sleep(1);")
16     (producer (call/cc consumer-continuation) (+ count 1))))
17
18 (define consumer
19   (lambda (producer-continuation)
20     (display "consumer eating ")
21     (display buffer)
22     (newline)
23     (set! buffer '()) ;accessing the shared resource
24     ;;(sleep 1) ;slowing things down a bit
25     (foreign-code "sleep(1);")
26     (consumer (call/cc producer-continuation))))
27
28 ;(trace producer consumer)
29
30 (producer consumer 0) ;getting the ball rolling
```

ProcLang/coroutines/producer-consumer.ref

- 1 producer making (tuna sandwich # 0)
- 2 consumer eating (tuna sandwich # 0)
- 3 producer making (tuna sandwich # 1)
- 4 consumer eating (tuna sandwich # 1)
- 5 producer making (tuna sandwich # 2)
- 6 consumer eating (tuna sandwich # 2)
- 7 producer making (tuna sandwich # 3)
- 8 consumer eating (tuna sandwich # 3)
- 9 producer making (tuna sandwich # 4)
- 10 consumer eating (tuna sandwich # 4)
- 11 producer making (tuna sandwich # 5)
- 12 consumer eating (tuna sandwich # 5)
- 13 producer making (tuna sandwich # 6)
- 14 consumer eating (tuna sandwich # 6)
- 15 producer making (tuna sandwich # 7)
- 16 consumer eating (tuna sandwich # 7)
- 17 producer making (tuna sandwich # 8)
- 18 consumer eating (tuna sandwich # 8)
- 19 producer making (tuna sandwich # 9)
- 20 consumer eating (tuna sandwich # 9)
- 21 producer making (tuna sandwich # 10)
- 22 consumer eating (tuna sandwich # 10)

ProcLang/coroutines/producer-consumer.py

```
1 # STUDENTS: Write producer-consumer.py so that it mirrors
2 # the behavior of producer-consumer.scm that we went over in class,
3 # subject to the following detailed requirements.
4 # The purpose of this assignment is to emulate coroutines using the
5 # Python generator construct.
6
7 # 1. buffer must be a Python variable bound to a string that is
8 #    global to this module.
9
10 # 2. producer must be a Python generator that yields control using
11 #    the yield operator. The producer takes ONE PARAMETER, a LIMIT
12 #    on the number of tuna sandwiches. It yields exactly that many
13 #    tuna sandwiches, 0 .. LIMIT-1, after which it returns.
14 #    Immediately prior to yielding a sandwich, producer must assign
15 #    a reference to that string into global variable buffer.
16
17 # 3. consumer must be a Python generator that yields control using
18 #    the yield operator. It must print the message, reassign global buffer
19 #    to an empty string, and yield a value of None whenever its next()
20 #    operator is invoked.
21
22 #    To append an integer value to a string such as
23 #    “producer making “, first convert the int to a string like this:
24 #        str(value)
25 #    then append that string.
26 #
27 #    Python’s “sleep” function is in the “time” library module.
28 #    Please mirror its use in producer-consumer.scm.
29
30 # 4. Write __main__ code for this test driver that does the following things:
31 # 4a. Make sure that the command line has two arguments (in sys.argv).
32 #    The first is the name of the program being run, and the second
33 #    must be an integer COUNT telling how many times to run the
34 #    producer-consumer interaction. Take a look at
35 #    compile_minimake/test_mmcompile.py from the last assignment to
36 #    see how to write a Python __main__ test driver and how to use
37 #    argv to access command-line arguments. Convert the command line
38 #    argument to an int using “int(argv[1])”.
39 # 4b. Construct one producer generator, passing the command line
40 #    COUNT as its argument.
41 # 4c. Construct one consumer generator.
42 # 4d. Iterate COUNT times, invoking producer.next() and consumer.next()
43 #    once per iteration. For this assignment you can ignore the values
44 #    returned from next(), since producer communicates values to
```

```
45 # consumer via global variable buffer, and consumer prints them out.
46
47 # 5. Add tests to the makefile so that running “gmake test” invokes
48 # producer-consumer.py with command line argument(s) so that it
49 # creates a producer-consumer.out file that matches
50 # producer-consumer.ref.
51 # Look at compile_minimake/makefile for an example makefile
52 # with some tests.
```

ProcLang/coroutines/makefile

```
1 # makefile for CSC 580 scheme directory -- builds a chicken file
2 # CSC580, Spring, 2009, Dr. Dale Parson.
3 # Updated to compile and run Liz Kalter's producer-consumer demo
4 # written using Scheme continuations, Spring, 2010, CSC 310.
5
6 all:    build
7
8 CHICKENLIB=/usr/local/lib
9 CHICKENBIN=/usr/local/bin/chicken
10
11 TARGET = producer-consumer
12 SOURCE = $(TARGET).scm
13 CSOURCE = $(TARGET).c
14
15 include ./makelib
16
17 build:  $(TARGET)
18
19 $(TARGET):$(SOURCE)
20                chicken $(SOURCE) -output-file $(CSOURCE)
21                gcc -I /usr/local/include $(CSOURCE) -L/usr/local/lib -lchicken -o
producer-consumer
22
23 clean:  subclean
24                /bin/rm -f *.out *.dif $(TARGET) producer-consumer.out producer-
consumer.dif $(CSOURCE)
25
26 # This rule is a test for the Scheme program, NOT for CSC310 assn. #4.
27 testscheme:$(TARGET)
28                LD_LIBRARY_PATH=/usr/local/lib ./producer-consumer
29
30 # STUDENT:
31 # ADD SOME TESTS BELOW AS REQUIRED BY THE ASSIGNMENT HANDOUT.
32 # See instructions in producer-consumer.py.
33 test:
```

Some examples of Python generator control.

```
>>> def genprint(limit):
...     print "starting to run genprint"
...     for i in range(0,limit):
...         print "genprint just before yield"
...         yield i
...         print "genprint just after yield"
...
>>> gp = genprint(4)
>>> gp.next()
starting to run genprint
genprint just before yield
0
>>> gp.next()
genprint just after yield
genprint just before yield
1
>>> gp.next()
genprint just after yield
genprint just before yield
2
>>> gp.next()
genprint just after yield
genprint just before yield
3
>>> gp.next()
genprint just after yield
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```